University of Southampton
Faculty of Engineering and Physical Sciences
Electronics and Computer Science

# Malware Analysis Economics – Investigating optimal techniques with limited resources

by

**Sandeep Mistry**

September 2019

Supervisor: Dr. Leonardo Aniello
Second Examiner: Prof. Stephen Gabriel

A dissertation submitted in partial fulfilment of
the degree MSc Artificial Intelligence

# Abstract

Malware are becoming an increasing threat in the modern world, and whilst those in the field of malware analysis have made great strides in improving detection and family classification accuracy, the economics of the analysis techniques have not been studied. This work focuses on Malware Analysis Economics, which studies the trade-offs encountered when attempting to maintain high classification accuracy and performance during malware analysis, along with the economic costs of the required equipment. This is the first full-length study into this novel research area. The largely unexplored area of malware category classification is undertaken on Windows executables, using static analysis methods including byte-level n-gram analysis and Portable Executable header analysis, and the time and space complexities of these methods are studied. New and original results on per sample feature extraction time and malware throughput are presented and used to construct an overview of how to economically maintain high accuracy. Finally, machine learning classifiers including XGBoost and Random Forest are used to obtain state-of-the-art classification results, which surpass those given in the relevant literature.

# Acknowledgements

I would like to thank my supervisor Dr. Leonardo Aniello for his support and guidance throughout the course of this project. I would also like to thank my parents and friends for their support and encouragement.

# Statement of Originality

I have read and understood the ECS Academic Integrity information and the University's Academic Integrity Guidance for Students.

I am aware that failure to act in accordance with the Regulations Governing Academic Integrity may lead to the imposition of penalties which, for the most serious cases, may include termination of programme.

I consent to the University copying and distributing any or all of my work in any form and using third parties (who may be based outside the EU/EEA) to verify whether my work contains plagiarised material, and for quality assurance purposes.

I have acknowledged all sources, and identified any content taken from elsewhere.

Some open-source code was partially used for byte-level n-gram analysis during this project. Source: https://github.com/kaanege/malware

I did all the work myself, or with my allocated group, and have not helped anyone else.

The material in the report is genuine, and I have included all my data/code/designs.

Some part of this was used in the submitted literature review for ELEC6211. I was given permission by my supervisor to do this.

My work did not involve human participants, their cells or animals.

# Contents

# Chapter 1

# Introduction

The modern world is becoming increasingly reliant on the internet and this presents opportunities for cyber-criminals to take advantage of, through the use of **mal**icious soft**ware** (malware).

The term malware encompasses a wide range of software, including viruses, worms, trojans, and ransomware [1]. The most common form of malware are viruses, which are usually hidden in executable files and infect the host when the file is executed - they are also capable of self-replication. Worms share similarities with viruses, such as self-replication, but operate independently. Trojans operate by tricking the user into executing them via deceit, i.e., disguised harmful e-mail attachments. Ransomware works by usually locking access to a user's personal data and demanding a monetary ransom in exchange for restoring access. A recent case of this was the WannaCry attack in 2017, which targeted Windows systems worldwide, encrypting user data and demanding a ransom in Bitcoin to decrypt it [2].

There exist three methods by which malware can be analysed: *static analysis* (where the program **is not** executed), *dynamic analysis* (where the program **is** executed), and *hybrid analysis* (which combines feature extraction methods from both static and dynamic analysis) [3]. Malware analysis can be performed on various representations of the malicious sample, including the raw executable itself, representations of the machine code and the assembly code.

Malware detection (malicious vs benign) and family classification have been covered extensively in the literature, but only a few works focus on malware category classification. A malware category is assigned according to general functionality [4], or more simply what the malware does, e.g., trojans disguise themselves as benign files to trick users into downloading and installing them.

Although much work has been done on malware classification using machine learning (ML) methods, the economics of these methods remain relatively unexplored, with only one study covering it to date, by Ucci et al. [5]. Malware Analysis Economics (MAE) focuses on the trade-offs encountered when attempting to maintain high accuracy and performance during malware analysis, along with the economic costs of the required equipment. MAE takes into account the time and space complexities of both feature extraction techniques and ML classifiers, as new malware are constantly being devel-

oped. Some techniques can result in excessively large feature spaces, leading to resource saturation. Features derived from dynamic analysis tend to be more valuable than those from static analysis, however since dynamic analysis requires running the sample, it leads to higher feature extraction time than static analysis. Parallelisation of analysis methods can help to reduce excessive execution times and memory consumption, however the additional equipment required to achieve this also has an economic cost - this must also be taken into account when choosing feature extraction methods. Dataset size can constitute a possible performance bottleneck, as the raw samples need to be physically stored, along with any additional data, e.g., raw code representations. Large datasets are generally needed to achieve more accurate ML models.

This work will focus on static analysis of malicious Windows executables and attempt to classify them into categories. It will aim to plug the present holes in malware analysis and inspire future work into category classification, to help those in related industries improve their malware analysis efficiency and efficacy. It will also aim to build a foundation on which to expand the new, novel concept of MAE and allow smaller groups, with limited to perform effective malware analysis.

## 1.1 Aims

The main aim of this project is to is to study MAE of multiple static analysis methods. The project goals are:

- Determine the static analysis method which gives highest classification accuracy.
- Given the static analysis methods used, determine the ML classifier which gives the highest category classification accuracy.
- Analyse feature extraction time between different static analysis methods.
- Determine the static analysis method which gives the highest malware throughput (number of malware samples analysed per second).
- Determine to what extent hardware configuration affects feature extraction time and malware throughput.
- Use the obtained results to construct an overview of how to maintain high classification accuracy whilst balancing economic costs.

## 1.2 Dissertation Structure

This dissertation will have the following structure: Chapter 2 will give a brief overview of Windows executables and their structure, the ML algorithms which will be used

to classify malware, a review of related work in the area of malware analysis and an explanation of the motivation behind this particular work. Chapter 3 will cover project management, including planning, progress, time management, and contingency planning. Chapter 4 will describe in detail the approach used in this work. Chapter 5 will cover implementation details, including dataset acquisition, preprocessing, and the testbed used. Chapter 6 will discuss the experiments undertaken, along with the results of these experiments and an explanation of the metrics used to judge performance. Finally, Chapter 7 will present the conclusions of the project and suggest future work which could help to build upon this research area.

# Chapter 2

# Background

This chapter will discuss the Portable Executable (PE) file structure, the ML algorithms used in this project, related work undertaken in the field and the motivation for this work.

## 2.1 PE File Format

The PE file format is the standard format for executables and DLLs in Windows systems [6]. PE files contain various fields at pre-defined locations, which contain vital information about them. The PE file structure is shown in Figure 2.1 [7].

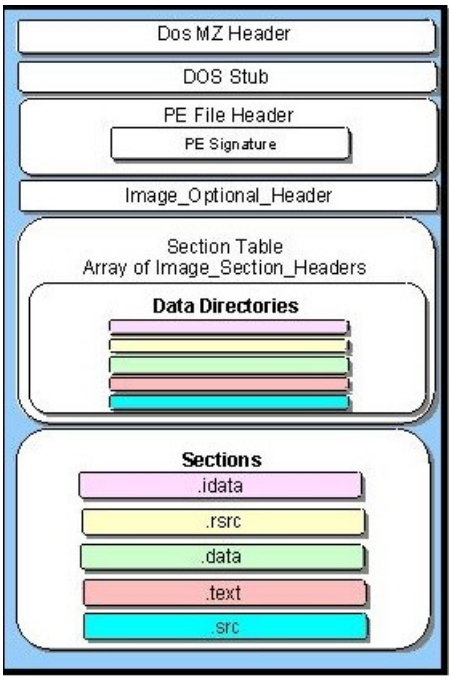



Figure 2.1: PE file structure

The DOS header allows DOS to recognise the file as an executable, whereas the DOS stub simply prints the string "This program cannot be run in DOS mode", which is displayed if one attempts to run the PE in an environment which does not support Win32. The PE file header contains information such as the number of sections in

the section table (which follows the header), and the target machine (CPU) type [8]. The Image_Optional_Header contains information such as operating system version and initial stack size. The Section Table gives information about each section in the succeeding sections; this includes their virtual size and the size of their raw data. Finally, the sections contain the main content of the PE, such as data, resources and imported functions.

Features extracted from the PE header and sections contain valuable information about the file and can be used in malware detection, similarity detection (e.g. families detection) and category classification.

## 2.2 Machine Learning Algorithms

ML algorithms have been used extensively in recent years to classify and detect malware [5]. These techniques can be split into three categories, which are summarised below [9], along with examples of each.

### 2.2.1 Supervised Learning

Supervised learning uses labelled training data. Examples include: Bayesian Networks, Support Vector Machines (SVMs), Random Forest, and k-Nearest-Neighbours (k-NN).

### 2.2.2 Unsupervised Learning

Unsupervised learning uses unlabelled data. Examples include: k-Means Clustering, Hierarchical Clustering, and Expectation Maximisation.

### 2.2.3 Semi-supervised Learning

Semi-supervised learning uses a combination of labelled and unlabelled training data, but has been rarely used in previous works. An example is Belief Propagation.

Five ML algorithms were used to classify malware in this project; the following sections will give a brief overview of them.

### 2.2.4 Random Forest

Random Forest consists of a large number of individual decision trees, which operate as an ensemble. The trees are largely uncorrelated and each one yields its own class prediction, with a majority vote deciding the winning class (Figure 2.2) [10]. Uncorrelated models can produce ensemble predictions that are more accurate than any of the individual predictions. Since the predictions made by individual trees are uncorrelated, so are their errors.

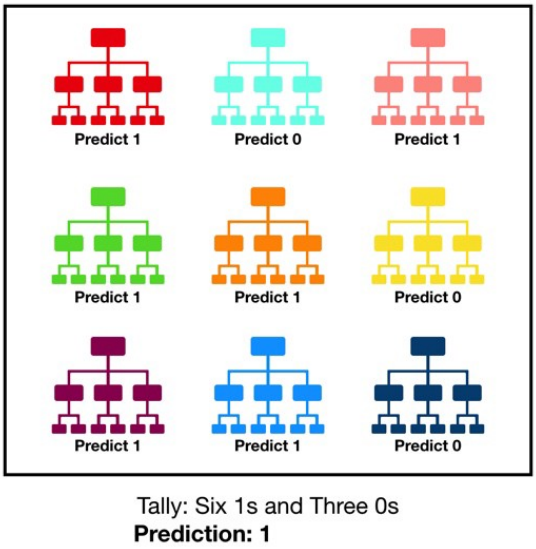



Figure 2.2: Random Forest algorithm.

### 2.2.5 XGBoost

XGBoost (eXtreme Gradient Boosting) also uses decision trees, however models are built sequentially by minimising the errors from previous models, whilst increasing (boosting) the influence of high-performing models (Figure 2.3) [11]. XGBoost improves upon regular Gradient Boosting by introducing system optimisation, in the form of parallelisation and tree pruning, specified by its `max_depth` parameter. It also features algorithmic enhancements such as regularisation and in-built cross-validation.

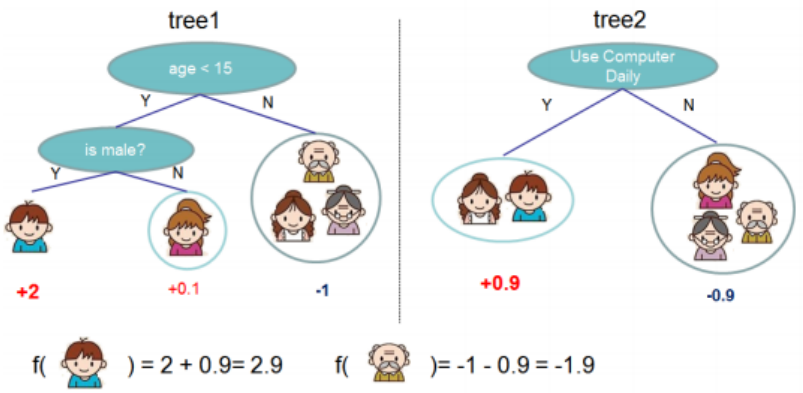



Figure 2.3: XGBoost algorithm.

### 2.2.6 Naive Bayes

The Naive Bayes classifier is a probabilistic model; it is based upon Bayes' theorem (2.1). Using Bayes' theorem, we can find the probability of an event, A, happening, given that another event, B, occurred. Here, B is the evidence and A is the hypothesis. We assume that the features are independent and thus do not affect each other - hence the term "naive" [12]. There are multiple types of Naive Bayes classifiers, including Multinomial, Bernoulli and Gaussian. In this project, the Multinomial Naive Bayes classifier was used, as it perform better on multi-category classification, compared with its Bernoulli and Gaussian counterparts.

$$P(A \mid B) = \frac{P(B \mid A)\, P(A)}{P(B)} \tag{2.1}$$

### 2.2.7 k-NN

The k-NN algorithm works by calculating the Euclidean distance between a single data point and all data points in a dataset. We select a specific number, $k$ data points closest to the single data point and then classify using a majority voting system (Figure 2.4) [13].

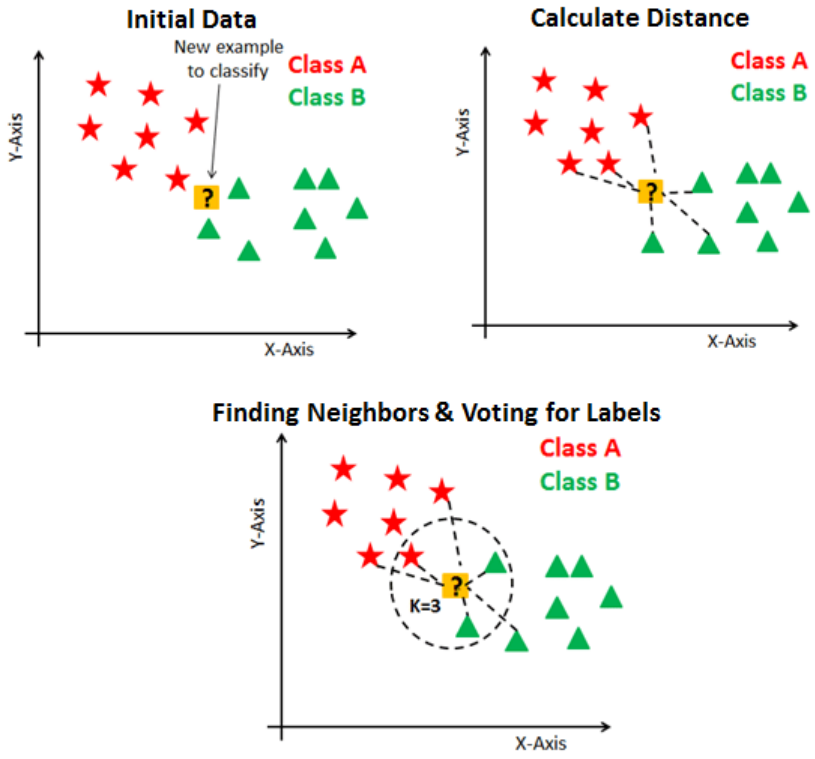



Figure 2.4: k-NN algorithm. For $k = 3$, we classify the new example as Class B.

### 2.2.8 Multilayer Perceptron

The Multilayer Perceptron (MLP) is a type of artificial neural network, consisting of at least three layers of nodes: an input layer, a hidden layer and an output layer [14] (Figure 2.5). Each input has an associated weight, which is then fed into a single or multiple hidden layers, where more complicated features can be learnt. Based on the features it has learnt, then network then makes a prediction. The power of an MLP lies in the backpropagation algorithm, where we run through the network in reverse and repeatedly adjust the weights to minimise the error between the actual output and the desired output.

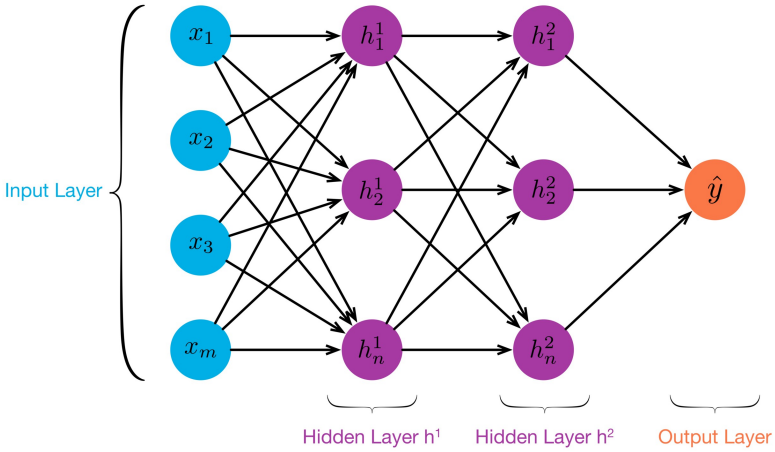



Figure 2.5: MLP with two hidden layers.

## 2.3 Related Work

There has been a wealth of work undertaken in the in the field of malware analysis using ML algorithms. This section will review a selection of them.

Banin and Dyrkolbotn [4] performed malware category classification on a dataset comprising of 952 samples balanced over 10 categories (Backdoor, PWS (Trojan that steals passwords), Rogue, Trojan, Trojandownloader, Trojandropper, Trojanspy, Virtool, Virus and Worm). They performed n-gram analysis on memory access patterns to classify the samples. They obtained more than 6M features from the dataset and noted that such data was too big to use in general-purpose ML libraries. To combat this, the features were ranked based on Information Gain (a quality measure based on class entropy and class conditional entropy, given the value of an attribute). They used the top ranked 5, 10, 15, 30 and 50,000 features to study classification performance dependency on the number of features. Correlation-based feature selection was used to obtain the 29 best features. The following ML classifiers were chosen: k-NN, Random Forest, Decision

Trees (J48), SVM, Naive Bayes and Neural Network. A joint peak accuracy of 66.80% was obtained using Random Forest and k-NN on the 29 best features, but Random Forest also managed to achieve this accuracy on the top 30,000 features as well. Poor performance was observed for the Naive Bayes classifier, obtaining a peak accuracy of 49.80% on the top 29 features.

Shalaginov et al. [15] performed category classification on a dataset of 328K PEs with 35 categories. They used Neuro-Fuzzy methods (a combination of neural networks and fuzzy logic), obtaining a peak accuracy of 26.27%. They noted that the quality and utility of the extracted features have a crucial influence on the model's performance and that additional analysis of the PE headers was required for this, indicating that feature quantity alone was not enough for accurate category classification.

Ahmadi et al. [16] used static analysis to extract various features from the hex dumps contained in the 20K sample Microsoft Malware Challenge dataset, including byte-level n-gram analysis. They noted that feature extraction and classifier training can be time consuming, when extracting complex features and the dataset size is large. The 2-gram category contained 65K features per sample and took 10213 seconds in total to extract. As the number of extracted features grows exponentially as $n$ is increased, they noted that the time frame required to extract 3-gram and 4-gram features was excessive. It should be noted that Microsoft chose to remove the headers from hex dumps of the samples, to ensure sterility and so the authors would have experienced a loss of vital information and lower feature extraction time, since they did not have the full hex dump for each sample. They used XGBoost and achieved an accuracy of 99.77% in malware family classification, with a very low log loss of 0.0096 on the training set.

Lin et al. [17] also showed that the time complexity and feature dimensionality (hence space complexity) increased exponentially as $n$ was increased (Figure 2.6) - increasing $n$ lead to higher accuracy. They used feature selection methods to decrease data dimensionality, leading to a decrease in analysis time complexity, and ultimately meeting the online requirement of collecting malware behaviour every minute. If these feature selection methods are employed, static analysis using n-grams becomes computationally cheaper, making it a more economically attractive option.

Baldangombo1 et al. [18] performed static analysis on a large 247,348 sample dataset of Windows programs in PE file format. They extracted key features from the each sample, including all the header information, DLL names, and API function names called within those DLLs. They noted that many samples had far too many features for learning schemes to handle and that the overwhelming majority were either irrelevant or redundant. They thus employed feature selection techniques, including ranking the top features within the header and only using them for classification. This method will be used in the work, to reduce total feature extraction time.

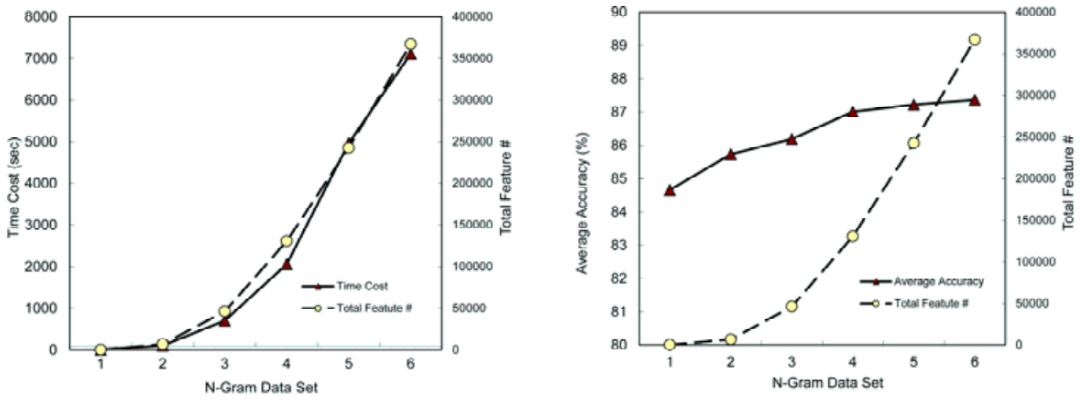



Figure 2.6: Relationship between time cost, total feature count and classification accuracy.

They used SVM, J48 and Naive Bayes classifiers on a machine with and Intel Pentium Core 2 Duo 2.33 GHz processor with 2GB RAM. No information on feature extraction time is given in this work, and it is possible that the number of features used was decreased excessively, to accommodate for the low compute power of their setup. To account for this, the project will perform experiments using multiple platforms with varying compute power.

Dahl et al.[19] used an NVIDIA Tesla C2075 to train a Neural Network (NN) on a data set with over 2.6 million samples, taking roughly three hours. Whilst this provided a peak detection accuracy of 99.58%, along with a relatively short training time, high-performance GPUs present a large monetary overhead cost which could prove to be a limiting factor in cases where this level of equipment is not available. Non-NN methods can classify malware within adequate time constraints only using CPUs, thereby appearing more economically attractive.

A key disadvantage of static malware analysis is that it can be evaded by *obfuscation, packing, or encryption*, and so the use of dynamic analysis can become necessary. Bayer et al. [20] suggested a method which exploited the fact that many new malware were simply variants of a select few malware, a technique known as *polymorphic reproduction*. They were able to avoid a full dynamic analysis in 25.25% of cases, saving 190.8 hours of analysis time. Dynamic analysis was required here because execution of the program had to detect if it was a polymorphic variant of a previously analysed program, which is not possible with static analysis. Therefore, more expensive dynamic analysis may be required to maintain high detection accuracy, even if n-grams methods are used in conjunction with feature selection techniques.

Kawaguchi and Omote [21] used dynamic analysis to extract API strings from malware samples and recorded an initial classification time of 90 seconds per sample, using methods including SVM, Random Forest and kNN - an average accuracy of over 70%

was found for all classifiers, whilst the Random Forest classifier alone recorded over 80%. Details of their experimental setup were not given however, and so it is difficult to extrapolate their classification time.

## 2.4 Motivation

Much of the related work covered has focused on malware detection and family classification, but category classification has not been not been explored as thoroughly. It is much harder than the two former classification methods, since the categories are too general and the importance of the numerical features is thus not high [15] - there is also a large overlap between classes, making classification even more difficult. This gave a strong motivation to focus on categories. Feature extraction times and feature space sizes of static analysis methods have also not been covered much in the related work and are a core research area here. The concept of malware throughput (the number of malware analysed per second) is not present much in literature and so this was included in the study. It was hoped that addressing these holes in the literature would help to produce an original contribution to the field of malware analysis.

# Chapter 3

# Project Management

This chapter will describe the planning, time management and contingency planning involved in the project.

The project began with an initial three week in-depth literature review, where a large volume of related literature was analysed and also open-source implementations of static malware analysis in Python were studied. This period laid out a foundation of knowledge on which it was possible to build a good understanding of current static analysis techniques and also what would not be possible to implement, e.g. dynamic analysis in the given time frame.

Difficulty was encountered in obtaining a suitable dataset. Indeed, the three largest proposed datasets (Microsoft Malware Challenge dataset, Endgame Malware BEnchmark for Research (EMBER) and Palo Alto Networks Ngrams dataset) proved to be unsuitable because they did not contain the raw binaries of the samples, meaning that feature extraction time could not be studied with them. Whilst the Microsoft dataset did contain hex dumps of 20K samples which could be used for n-grams analysis, the header of each sample had been removed, meaning they were not realistic examples of malware that could be encountered in the wild. Following this, a contingency plan of creating an original dataset with samples collected from VirusShare.com, using an open-source implementation and classifying them using VirusTotal reports would be enacted if a suitable dataset could not be found.

Due to the difficulty of implementing many feature extraction methods witnessed in the research undertaken in the initial three week period, it was decided that at most four methods would be investigated, allowing for more comprehensive testing in the restricted time frame. Although more methods could have been implemented, they would most likely not have had much depth, hence the initial planning helped to ensure that valuable results would be obtained.

It was not feasible to run intensive feature extraction methods for an extended period of time on a personal computer and so use of the University's high performance computing cluster, Lyceum was attempted. However, it was later considered that the nature of the project should restrict computing power and Lyceum was deemed too powerful. Instead, it was decided that Google Cloud Platform would instead be used, due to the availability of many different hardware configurations of varying compute power. Google

also offered $300 worth of free credits, giving the added benefit of not requiring the pre-allocated project budget of £200. It was also hypothesised that use of a less powerful setup would highlight excessive feature extraction time and memory usage.

Figure 3.1 shows a Gantt chart of the project and the original Gantt chart given before commencement of the project is given in the appendix. The key takeaways when comparing the two charts are the major increase in time it took to obtain a suitable dataset, a reduction in the number of static analysis techniques investigated, and an increase in time given to initial testing on sub-samples of the dataset.

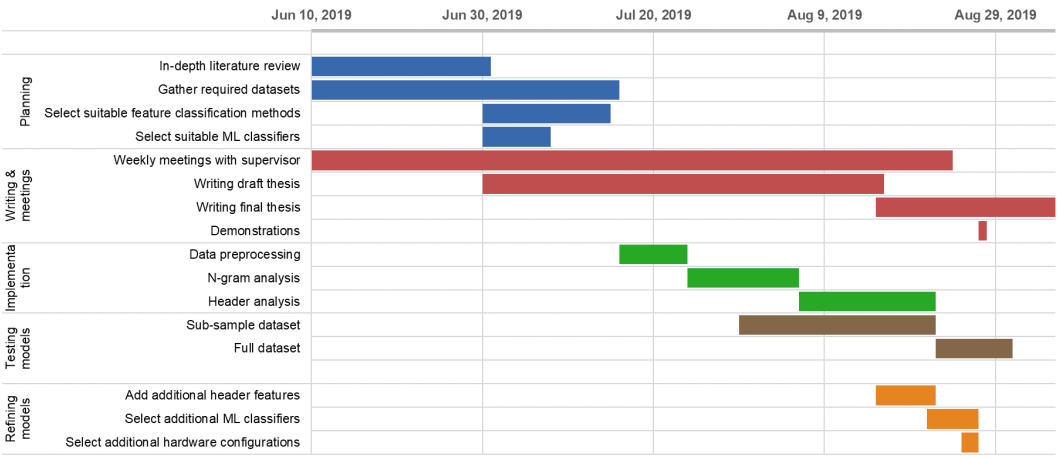



Figure 3.1: Gantt chart showing progress during the project.

# Chapter 4

# Approach

This chapter will describe and present an intuitive overview about the approaches taken in this work.

## 4.1 Byte-Level N-gram Analysis

N-grams are a concept commonly found in the field of natural language processing, however they have powerful applications in malware analysis [16]. They consist of contiguous sequences of $n$ words in a given corpus. Take the sentence "This project focuses on MAE" - the 1-grams (or unigrams) of this sentence would be "This", "project", "focuses", "on" and "MAE". The 2-grams (or bigrams) would be "This project", "project focuses", "focuses on" and "on MAE". We assign probabilities to sequences of words of the format in (4.1).

$$P(W \mid H) \tag{4.1}$$

Where $W$ is a word and $H$ is a history of words. The probability function can be estimated by counting the relative frequencies of $H$ being followed by $W$. For example given our original sentence, $P(MAE \mid This\ project\ focuses\ on)$ is 1, since the words "This project focuses on" immediately precede the word "MAE". Similarly, $P(project \mid This\ project\ focuses\ on)$ is 0.

In the case of malware analysis, the same approach can be taken on the byte-code of malware samples. N-gram analysis is done on the hex dumps (hexadecimal view of the machine code) of samples, since they contain vast amounts of information about the nature of the sample and are much easier to analyse than binary code. The hex dump format is shown in 5.2.2. As $n$ is increased, the number of features increases exponentially, making it difficult to perform n-gram analysis on large datasets with high $n$ values. Because of this, a maximum value of $n = 3$ was investigated in this project.

## 4.2 Header Analysis

As described in section 2.1, the header of a PE contains valuable information about its operations. A large number of features (115) were extracted from the headers of each

sample, as proposed by Laurenza et al. [22]. A meta-transformer for selecting features based on importance weights was then applied to the extracted features and the top 25 were used for final training and testing. A selection of the features used are described in the next few subsections.

### 4.2.1 Section Entropy

Entropy is a measure of randomness, and in the case of PE sections it is used as a measure of how disordered the bytes are in the PE [7]. We use Shannon's formula (4.2) to calculate entropy.

$$\sum_{i} p_i \, log \, \frac{1}{p_i} \tag{4.2}$$

Where $i$ is the event with probability $p_i$. The result of this equation is always between 0 and 8. The entropies of the most frequent sections (`.idata`, `.rdata`, `.data`, `.rsrc` and `.reloc`) were calculated.

### 4.2.2 Imported API Counts

Application programming interface (API) calls tell systems to perform certain operations and we can count the number of times they are asked to perform these operations. The number of API counts can give a lot of information about what the PE is designed to do [7]. The imported API counts from various common dynamic linked library (dll) files were used, along with whether specific API calls were present. The full list of these is given in the appendix.

### 4.2.3 Miscellaneous Features

Other miscellaneous features covered in related works on PE header analysis [16] were included. These included the file size, number of sections, whether the PE is 32-bit or not and the creation year.

# Chapter 5

# Implementation

This chapter will describe the dataset used, preprocessing performed on the data and the software and hardware configurations used during the project.

## 5.1 Dataset

The dataset[1] used in this project comprised of 63,628 malicious PE files collected from VirusShare and VX Heaven; it was created for use in an undergraduate project at Heriot-Watt University [23]. It comprises solely of Windows-based malware samples, as it is the most ubiquitous operating system and hence the target of a majority of malware. The samples from VXHeaven are older than those from VirusShare, adding an element of diversity to the dataset. Table 5.1 shows the dataset sources and composition.

| Collection | Samples Used | Percentage of Dataset |
|---|---|---|
| **VirusShare_00188** | 21,149 | 33.24% |
| **VirusShare_00189** | 34.056 | 53.52% |
| **VX Heaven Collection** | 8,423 | 13.24% |

Table 5.1: Dataset sources.

The dataset contained an even distribution of 10,000 samples per category. The only exceptions were the generic category, which contained 9,030 samples and the unclassified category, which contained 4,598 samples. The samples were classified by identifying the most frequently occurring string in their VirusTotal reports, i.e., "backdoor", "adware", etc. The Microsoft Malware Encyclopedia was used to classify samples when the VirusTotal reports contained an equal number of strings classifying them as different types of malware. Table 5.2 shows the malware category distribution of the dataset.

[1] Available at https://github.com/naisofly/Static-Malware-Analysis

| Malware Type | Number of Samples |
|---|---|
| **Adware** | 10,000 |
| **Backdoor** | 10,000 |
| **Generic** | 9,030 |
| **Trojan** | 10,000 |
| **Virus** | 10,000 |
| **Worm** | 10,000 |
| **Unclassified** | 4,598 |

Table 5.2: Dataset category distribution.

## 5.2 Data preparation

### 5.2.1 Labelling

Labelling of the samples was based off of the provided ground truth in the dataset. Each sample was renamed to its unique MD5 hash (Listing 5.1) and assigned a numerical value between 0-6 to signify its class. Linux commands were used for all data preparation tasks.

Listing 5.1: MD5 sample renaming code

```
mv filename (md5sum filename | awk '{ print 1 }')
```

### 5.2.2 Hex dumps

Hex dumps were extracted from each sample using the command `od` (Listing 5.2) into a standard format (Listing 5.3). In the hex dumps, the first seven numbers of each line denote the memory address (specific memory locations used by hardware and software) and the remaining two-digit hexadecimal numbers represent the byte-code. They are needed because they contain vital information about the nature of the PE being studied [7].

Listing 5.2: Hexdump extraction code

```
od -t x1 -v filename | head -n -1 | tr a-f A-F > filename.bytes
```

Listing 5.3: Hexdump format

```
0000000 4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00
0000020 B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00
0000040 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000060 00 00 00 00 00 00 00 00 00 00 00 00 F8 00 00 00
```

## 5.3 Testbed

To ensure reproducibility of the experimental results, all experiments were ran on Google Cloud Platform (GCP). Three different hardware combinations[2] were used, whilst the software specifications were kept constant, as it was unlikely that any significant differences in experimental results would be observed if they were changed. The open-source code used in this work is available on a public GitHub repository[3].

### 5.3.1 Software Specifications

- OS: Ubuntu 18.04 LTS.
- Language: Python 3.6.8.
- Libraries: `matplotlib`, `numpy`, `pandas`, `pefile`, `pickle`, `scikit-learn`, `scipy`, `xgboost`, `time`, `hashlib`.

### 5.3.2 Hardware Specifications

**Hardware Specification A (Spec. A)**

- Machine type: 4 vCPUs with 15 GB of memory (n1-standard-4).
- CPU platform: Intel Broadwell.
- Boot disk: 200GB Standard persistent disk (HDD).

**Hardware Specification B (Spec. B)**

- Machine type: 8 vCPUs with 30 GB of memory (n1-standard-8).
- CPU platform: Intel Haswell.
- Boot disk: 200GB SSD persistent.

**Hardware Specification C (Spec. C)**

- Machine type: 16 vCPUs with 14.4 GB memory (n1-highcpu-16)
- CPU platform: Intel Broadwell.
- Boot disk: 200GB SSD persistent.

[2]https://cloud.google.com/compute/docs/machine-types
[3]https://github.com/sandeep-mistry/MSc-Project-Malware-Analysis-Economics

# Chapter 6

# Experiments and Results

This chapter details the experiments undertaken and the results of these experiments. Preliminary experiments were undertaken on a small 1000 sample balanced sub-dataset, to test feature extraction time and resource utilisation.

## 6.1 Experiments

### 6.1.1 Byte-Level N-gram Analysis

Byte-level n-gram analysis was performed on the body of each sample's hex dump, yielding a list of dictionaries, in which each dictionary entry was of the form `byte value:byte frequency`. These entries were then vectorised, using `sklearn`'s `DictVectorizer` [24] class and transformed into sparse matrices. Sparse matrices were used because they used less memory than standard `numpy` arrays. They were then split into an 80:20 train:test ratio and classified using the classifiers mentioned in section 2.2. SVMs were not used because they tend to perform poorly on large datasets.

1-gram, 2-gram and 3-gram analysis was undertaken, however higher $n$ values were not used due to the exponential increase in the number of unique n-grams encountered as $n$ is increased [25]. It was hypothesised that feature extraction time would become excessive for $n$ values larger than 3. Sparse matrices were not used for 1-gram analysis, as the feature count here was not excessive.

For 2-gram and 3-gram analysis, memory usage and classification time became excessive when all features were used, and so only the top $k$ most frequently occurring n-grams were considered. An upper limit of $k = 500$ was considered for 2-gram analysis, as done in [26]. Any values higher than this did not give economical feature extraction times. For 3-gram analysis, an upper limit of $k = 100$ was considered, again due to excessive feature extraction times with higher values. Although this solved the memory issue, it added to total feature extraction time, as the dictionaries for each sample needed to be ordered by byte frequency. Since only the top $k$ features were considered, significant amounts of information were discarded, thus decreasing average classification accuracy.

### 6.1.2 Header Analysis

The Python library `pefile` was used to extract header features directly from each malicious sample, including imported API counts, directory entry imports and section entropies. The full list of 115 features is given in the appendix. These features were ranked in order of their importance towards classification, using `sklearn`'s `SelectFromModel` [24] class, and the only the top 25 features were retained.

For some samples, certain features were not able to be extracted, particularly section entropies and so in these cases the mean value for all samples was imputed. The `sklearn` function `StandardScaler` was used to standardise features by mean-centring and scaling to unit variance. This was done because many classifiers work best when features closely resemble normally distributed data. `StandardScaler` uses the Z-score formula (5.1).

$$z = \frac{x - \mu}{\sigma} \tag{6.1}$$

Where $x$ is the current value, $\mu$ is the mean value of the feature column $x$ is in, $\sigma$ is the standard deviation of the column and $z$ is the normalised value. `StandardScaler` was not used for the Multinomial Naive Bayes classifiers because it cannot accept negative feature vectors.

## 6.2 Metrics Used

In addition to classification accuracy, 10-fold cross-validation was implemented to test the generalisation abilities of the classifiers used. To test model performance at all classification thresholds, the ROC (Receiver Operating Characteristic) curve can be plotted - it has two parameters, True Positive Rate (TPR) and False Positive Rate (FPR).

$$TPR = \frac{TP}{TP + FN} \tag{6.2}$$

$$FPR = \frac{FP}{FP + TN} \tag{6.3}$$

Where $TP$ is a true positive, $FP$ is a false positive, $FN$ is a false negative and $TN$ is a true negative. A ROC curve plots $TPR$ against $FPR$ at all classification thresholds. The AUC (Area Under ROC Curve) is the two-dimensional area underneath the ROC curve and provides an measure of performance across all possible classification thresholds - it is a value between 0 and 1, where being closer to 1 indicates better performance. In a multi-class model, we plot $N$ ROC curves, where $N$ is the number of classes, using a one vs. all methodology. The AUC score was hence used to test classification performance across all classes. Figure 5.1 [27] shows a typical ROC curve and the AUC is shown by the shaded area.

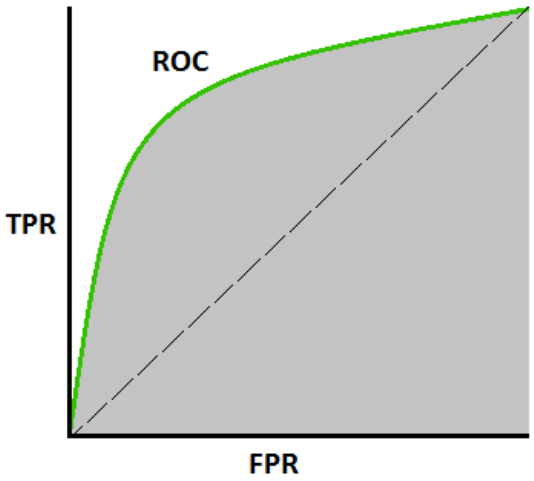



Figure 6.1: A typical ROC curve (AUC shown by shaded area).

Feature extraction time was measured in seconds and hardware cost in USD per hour. Total malware throughput, i.e. the number of malware samples analysed per second was also measured. Hardware costs were minimised by operating on servers in the cheapest locations available (us-west1-b, us-east1-b and us-east4-c), according to GCP's pricing standards. Latency due to server location was negligible.

## 6.3 Results

### 6.3.1 Header Analysis

The Random Forest and XGBoost classifiers yielded the highest classification accuracies with 83.29% and 83.53% respectively. A similar level of accuracy was found when training and testing across the whole dataset, with mean cross-validation accuracies of 82.26% and 82.43% respectively, whilst mean AUC scores were 0.89 for both algorithms. K-NN gave a slightly lower accuracy of 77.01% and AUC of 0.86, whilst MLP's accuracy and AUC were significantly lower, at 63.87% and 0.75 respectively. Naive Bayes performed very poorly, giving an accuracy of 28.26% and AUC of 0.56, indicating that it is not a good choice of classifier for malware category classification. Table 6.1 shows the results from using header analysis.

| Classifier | Accuracy (%) | Mean Cross-Val (%) | Cross-Val Variance (%) | Mean AUC |
|---|---|---|---|---|
| **Random Forest** | 83.29 | 82.26 | 0.34 | 0.89 |
| **Naive Bayes** | 28.26 | 28.33 | 0.30 | 0.56 |
| **XGBoost** | 83.53 | 82.43 | 0.30 | 0.89 |
| **k-NN** | 77.01 | 76.86 | 0.43 | 0.86 |
| **MLP** | 63.87 | 64.34 | 0.65 | 0.75 |

Table 6.1: Header analysis accuracies.

Differences in feature extraction time were observed between the hardware configurations used (Table 6.2). Whilst Spec. B offered a 37.50 ms decrease per sample, Spec. C only gave a 0.82 ms improvement on this, despite Spec. C having double the amount of cores than Spec. B. It was unlikely that Spec. B's higher memory caused this, as memory usage was generally low during feature extraction. This indicates that simply supplying more resources for header analysis may not give significant decreases in feature extraction time and that high-performance hardware is unnecessary.

| Hardware Specification | Feature Extraction Time Per Sample (ms) |
|---|---|
| **Spec. A** | 105.66 |
| **Spec. B** | 68.16 |
| **Spec. C** | 67.34 |

Table 6.2: Header analysis extraction times.

### 6.3.2 N-grams Analysis

1-gram analysis produced lower accuracies and AUC scores across all classifiers, compared with header analysis. The MLP in particular yielded a very low accuracy of 19.13% and AUC score of 0.52. This is paradoxical, because neural networks tend to perform better when they are fed more data and even 1-gram analysis had a higher feature count than header analysis in these experiments. Indeed, all classifiers were fed more data during n-grams analysis, yet their performance was worse and hence it is likely that the majority of this extra data was redundant.

| Classifier | Accuracy (%) | Mean Cross-Val (%) | Cross-Val Variance (%) | Mean AUC |
|---|---|---|---|---|
| **Random Forest** | 74.52 | 74.27 | 0.29 | 0.84 |
| **Naive Bayes** | 27.65 | 28.85 | 6.55 | 0.56 |
| **XGBoost** | 77.69 | 76.96 | 0.27 | 0.86 |
| **k-NN** | 67.84 | 67.42 | 0.42 | 0.81 |
| **MLP** | 19.13 | 20.53 | 4.09 | 0.52 |

Table 6.3: 1-gram analysis accuracies.

Although 1-gram analysis produced lower accuracies and AUC scores, feature extraction was noticeably quicker across all hardware specifications (Spec A. 70% faster, Spec. B 49% faster and Spec. C 25% faster). This decrease in extraction time, particularly the large decrease on the cheapest hardware configuration presents a major benefit to using 1-gram analysis over header analysis.

The results of 2-gram analysis were an improvement upon 1-gram analysis, particularly in the case of the MLP classifier, however a large variance was observed in the cross validation accuracies, indicating that the MLP does not perform consistently over the entire dataset. Increases in accuracy and mean AUC score came at the cost of an

| Hardware Specification | Feature Extraction Time Per Sample (ms) |
|---|---|
| **Spec. A** | 73.97 |
| **Spec. B** | 33.29 |
| **Spec. C** | 17.14 |

Table 6.4: 1-gram analysis extraction times.

increase in feature extraction time over all hardware configurations. Indeed, an increase of 470% was observed for Spec. A, 549% for Spec. B and 645% for Spec. C. These are very large increases in feature extraction time, considering that 2-gram analysis did not give major improvements upon 1-gram analysis for the higher-performing classifiers (3.02% for XGBoost and 4.33% for Random Forest). It must also taken into account the fact that 2-gram analysis only considered the top 500 most frequent n-grams and so a lot of information was discarded, however using all n-grams would have required hardware much superior to those used here and feature extraction time would have increased significantly due to the need to vectorise many more feature mappings.

| Classifier | Accuracy (%) | Mean Cross-Val (%) | Cross-Val Variance (%) | Mean AUC |
|---|---|---|---|---|
| **Random Forest** | 78.85 | 78.65 | 0.10 | 0.87 |
| **Naive Bayes** | 46.00 | 46.18 | 0.60 | 0.68 |
| **XGBoost** | 80.71 | 80.12 | 0.06 | 0.88 |
| **k-NN** | 68.62 | 68.32 | 0.16 | 0.81 |
| **MLP** | 48.39 | 42.04 | 23.47 | 0.68 |

Table 6.5: 2-gram analysis accuracies.

| Hardware Specification | Feature Extraction Time Per Sample (ms) |
|---|---|
| **Spec. A** | 347.68 |
| **Spec. B** | 182.90 |
| **Spec. C** | 110.56 |

Table 6.6: 2-gram analysis extraction times.

As only the top 100 most frequent n-grams were considered for 3-gram analysis, little improvement was seen in classification accuracy and AUC scores, however decreases in accuracy were observed in both XGBoost and Random Forest (2.09% and 0.58% respectively). There was considerable improvement in the MLP, however its results were still not in line with the top classifiers. Feature extraction time increased significantly, compared with 2-gram analysis (351% for Spec. A, 357% for Spec. B and 528% for Spec. C).

Figure 6.2 shows the average feature extraction time per sample and malware throughput for all hardware configurations. It can be observed that 1-gram analysis using Spec. C gives a significantly higher throughput than all other methods and that hardware con-

| Classifier | Accuracy (%) | Mean Cross-Val (%) | Cross-Val Variance (%) | Mean AUC |
|---|---|---|---|---|
| **Random Forest** | 78.27 | 78.08 | 0.06 | 0.87 |
| **Naive Bayes** | 50.82 | 51.53 | 0.58 | 0.71 |
| **XGBoost** | 78.92 | 78.84 | 0.23 | 0.87 |
| **k-NN** | 68.36 | 67.90 | 0.31 | 0.81 |
| **MLP** | 59.85 | 52.54 | 12.67 | 0.75 |

Table 6.7: 3-gram analysis accuracies.

| Hardware Specification | Feature Extraction Time Per Sample (ms) |
|---|---|
| **Spec. A** | 1221.06 |
| **Spec. B** | 653.01 |
| **Spec. C** | 584.09 |

Table 6.8: 3-gram analysis extraction times.

figuration has a clear impact on both feature extraction time and throughput. Figure 6.3 shows the accuracies for the top three classifiers across all feature extraction methods. Figure 6.4 shows the raw feature count (before feature selection techniques were employed) for each feature extraction method. It can be seen that the feature space becomes excessively large for 2-gram and 3-gram analysis, even when it is reduced by selecting the top $k$ most frequent n-grams.

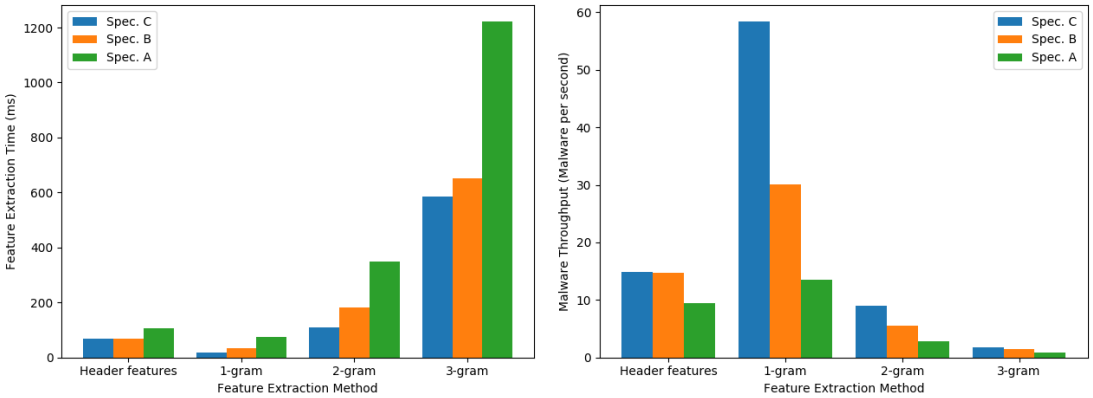
The accuracies reported here for all classifiers except XGBoost are higher than those reported by Banin and Dyrkolbotn [4] (this study did not use XGBoost), however they used a much smaller dataset of 952 samples, balanced across 10 categories. They achieved a joint peak accuracy of 68.80% using Random Forest and k-NN. This study also used more features (29) than were used by the top-performing feature extraction method used here (25). The top performing classifiers here were in agreement with this study (Random Forest and k-NN).

Much higher accuracies were record than those of Shalaginov et al. [15], who used neural networks and Neuro-Fuzzy methods. They achieved a peak accuracy of 26.47%. It should be noted that they used a much larger dataset (328K samples) than the one used here, but it consisted of 35 categories.

## 6.4 MAE Analysis

This section will address the aim of constructing an overview of MAE in the case of malware category classification.

As XGBoost gave the best performance across all feature extraction methods, it is recommended that this classifier be used when attempting to maintain high accuracy. In the case of header analysis, Random Forest is an adequate alternative, as the give near identical accuracies. Although 1-gram analysis gave the highest malware throughput,




(a) Feature Extraction Time per Sample

(b) Malware Throughput

Figure 6.2: Plots of average feature extraction time per sample and total malware throughput per second.

along with moderate accuracy, the resources required to achieve this may be excessive. Indeed, during all n-gram analyses, available resources were constantly saturated and GCP recommended increasing resources mid-way throughout the feature extraction and classification stages.

2-gram and 3-gram analysis, using the top $k$ (where $k \leqslant 500$) most frequent n-grams should be avoided, as only moderate improvements are observed at a very high cost of feature extraction time, resulting in low throughput.

Header analysis produces higher accuracies whilst using much less resources, and does not require high-performance hardware to maintain moderate throughput, as evidenced in Figure 6.2 where throughput between Spec. B and Spec. C were almost identical. It is recommended that this method be used when accuracy is favoured over high throughput. It is also recommended that feature selection techniques be employed on a small sub-sample of the dataset being used so that feature extraction time can be reduced for the entire dataset and higher throughput can be achieved, with negligible loss of accuracy. It is difficult to precisely quantify the increase in throughput, as different features may require more time to be extracted than others.

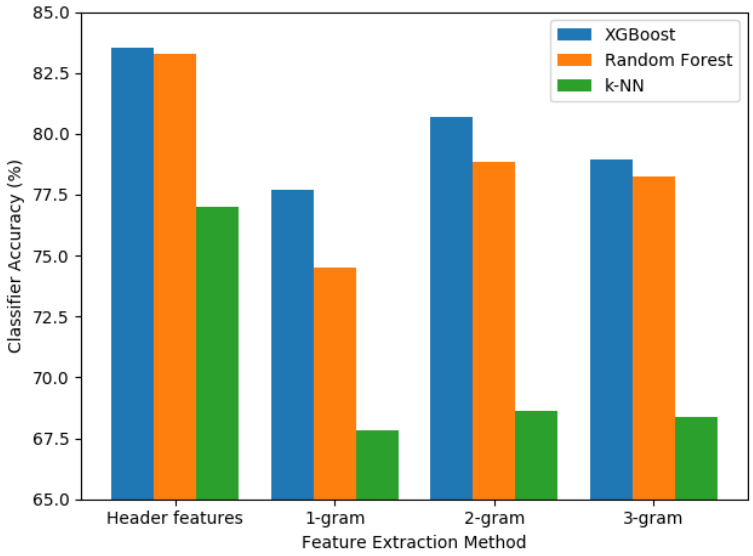



Figure 6.3: Accuracies for top three classifiers across all feature extraction methods.

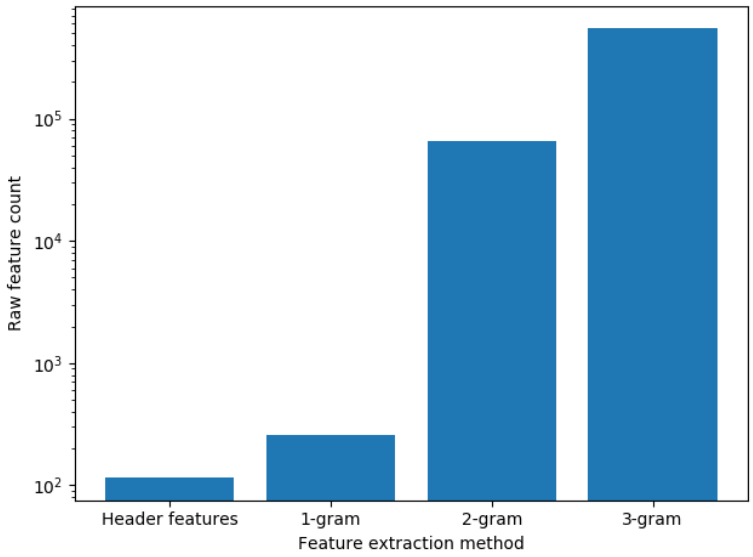



Figure 6.4: Raw feature count per sample for all feature extraction methods (in logarithmic scale).

# Chapter 7

# Conclusions and Future Work

In this work, the time and space complexities of multiple static analysis methods for malware analysis were analysed, along with comparisons between different ML classifiers for the largely unexplored area of malware category classification of PEs, and a foundation on which to expand the novel research area of MAE was presented. It was found that XGBoost yielded the highest classification accuracies of the chosen ML classifiers, with a peak accuracy of 83.53% using header analysis. N-gram analysis was only found to be economical for $N = 1$, using three different hardware configurations of varying computing power, and a peak throughput of 58 malware per second was achieved. Header analysis was found to be the optimal choice when accuracy was favoured along with moderate throughput. The results obtained in this work exceed the accuracies given in relevant literature, however further research should be conducted on industry-level datasets to test the applicability of the methods employed here to real-world scenarios.

Future work should explore a range of the $k$ most frequently occurring n-grams for byte-sequence n-grams analysis. Outside of header analysis, classification accuracy is heavily determined by the number of features, as demonstrated by Banin and Dyrkolbotn [4], however a higher $k$ value will significantly increase feature extraction time and give an unprocessable feature count if more computational resources are not provided. This would hence be an important area of MAE to study.

Few static analysis methods were studied in this work and there are a wide array available, including operation code (opcode) analysis and ASCII string lengths, as explored by Ahmadi et al. [16]. Future work should aim to research these methods in a similar manner to what was done in this project as they are much less resource intensive than byte-level n-gram analysis.

Feature quality, rather than quantity was found to be the driving factor in classification accuracy, as shown by the high-performance of header analysis. Hence further work should be undertaken into the development of new variants of current features which could differentiate between different malware categories with much lower time and space complexities than existing, intensive static analysis methods.

# Bibliography


[1] Michael Sikorski and Andrew Honig. *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. No Starch Press, San Francisco, CA, USA, 1st edition, 2012.

[2] Savita Mohurle and Manisha Patil. A brief study of wannacry threat: Ransomware attack 2017. *International Journal of Advanced Research in Computer Science*, 8(5), 2017.

[3] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. A survey on automated dynamic malware-analysis techniques and tools. *ACM Computing Surveys - CSUR*, 44:1–42, 02 2012.

[4] Sergii Banin and Geir Olav Dyrkolbotn. Multinomial malware classification via low-level features. *Digital Investigation*, 26:S107–S117, 2018.

[5] Daniele Ucci, Leonardo Aniello, and Roberto Baldoni. Survey of machine learning techniques for malware analysis. *Computers & Security*, 2018.

[6] Matt Pietrek. Peering inside the pe: A tour of the win32 portable executable file format. 1994.

[7] Malware researcher's handbook. `https://resources.infosecinstitute.com/2-malware-researchers-handbook-demystifying-pe-file`. Accessed: 04-09-2019.

[8] Pe format - windows applications. `https://docs.microsoft.com/en-us/windows/win32/debug/pe-format#machine-types`. Accessed: 04-09-2019.

[9] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, Berlin, Heidelberg, 2006.

[10] Understanding random forest - towards data science. `https://towardsdatascience.com/understanding-random-forest-58381e0602d2`. Accessed: 25-08-2019.

[11] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794. ACM, 2016.

[12] Matthew Kirk. *Thoughtful Machine Learning - A Test-Driven Approach.* O'Reilly, 2015.

[13] Knn classification using scikit-learn. `https://www.datacamp.com/community/tutorials/k-nearest-neighbor-classification-scikit-learn`. Accessed: 25-08-2019.

[14] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* The MIT Press, 2016.

[15] Andrii Shalaginov, Lars Strande Grini, and Katrin Franke. Understanding neuro-fuzzy on a class of multinomial malware detection problems. In *2016 International Joint Conference on Neural Networks (IJCNN)*, pages 684–691. IEEE, 2016.

[16] Mansour Ahmadi, Dmitry Ulyanov, Stanislav Semenov, Mikhail Trofimov, and Giorgio Giacinto. Novel feature extraction, selection and fusion for effective malware family classification. In *Proceedings of the sixth ACM conference on data and application security and privacy*, pages 183–194. ACM, 2016.

[17] Chih-Ta Lin, Nai-Jian Wang, Han Xiao, and Claudia Eckert. Feature selection and extraction for malware classification. *J. Inf. Sci. Eng.*, 31(3):965–992, 2015.

[18] Usukhbayar Baldangombo, Nyamjav Jambaljav, and Shi-Jinn Horng. A static malware detection system using data mining methods. *arXiv preprint arXiv:1308.2831*, 2013.

[19] George E Dahl, Jack W Stokes, Li Deng, and Dong Yu. Large-scale malware classification using random projections and neural networks. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 3422–3426. IEEE, 2013.

[20] Ulrich Bayer, Engin Kirda, and Christopher Kruegel. Improving the efficiency of dynamic malware analysis. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 1871–1878. ACM, 2010.

[21] Naoto Kawaguchi and Kazumasa Omote. Malware function classification using apis in initial behavior. In *2015 10th Asia Joint Conference on Information Security*, pages 138–144. IEEE, 2015.

[22] Giuseppe Laurenza, Leonardo Aniello, Riccardo Lazzeretti, and Roberto Baldoni. Malware triage based on static features and public apt reports. In *International Conference on Cyber Security Cryptography and Machine Learning*, pages 288–305. Springer, 2017.

[23] Naiyarah Hussain. *Malware Analysis & Detection Using Machine Learning Classifiers*. PhD thesis, Heriot-Watt University, 2016.

[24] Scikit-learn. `https://scikit-learn.org/stable/`. Accessed: 04-09-2019.

[25] Richard Zak, Edward Raff, and Charles Nicholas. What can n-grams learn for malware detection? In *2017 12th International Conference on Malicious and Unwanted Software (MALWARE)*, pages 109–118. IEEE, 2017.

[26] Abdurrahman Pektaş, Mehmet Eriş, and Tankut Acarman. Proposal of n-gram based algorithm for malware classification. In *The Fifth International Conference on Emerging Security Information, Systems and Technologies*, pages 7–13, 2011.

[27] Understanding auc roc curve towards data science. `https://towardsdatascience.com/understanding-auc-roc-curve-68b2303cc9c5`. Accessed: 01-09-2019.

# Appendix

| AUC Scores | | | | | | | |
|---|---|---|---|---|---|---|---|
| **Classifier** | **Adware** | **Backdoor** | **Generic** | **Trojan** | **Virus** | **Worm** | **Unclassified** |
| **Random Forest** | 0.93 | 0.95 | 0.81 | 0.83 | 0.95 | 0.96 | 0.79 |
| **XGBoost** | 0.93 | 0.95 | 0.81 | 0.84 | 0.95 | 0.96 | 0.80 |
| **k-NN** | 0.94 | 0.92 | 0.91 | 0.77 | 0.76 | 0.90 | 0.79 |
| **MLP** | 0.89 | 0.86 | 0.86 | 0.63 | 0.68 | 0.86 | 0.50 |
| **Naive Bayes** | 0.62 | 0.53 | 0.50 | 0.50 | 0.67 | 0.61 | 0.50 |

Table 7.1: Header analysis AUC scores.

| AUC Scores | | | | | | | |
|---|---|---|---|---|---|---|---|
| **Classifier** | **Adware** | **Backdoor** | **Generic** | **Trojan** | **Virus** | **Worm** | **Unclassified** |
| **Random Forest** | 0.93 | 0.87 | 0.75 | 0.76 | 0.89 | 0.93 | 0.77 |
| **XGBoost** | 0.93 | 0.90 | 0.77 | 0.79 | 0.91 | 0.94 | 0.79 |
| **k-NN** | 0.90 | 0.83 | 0.74 | 0.71 | 0.79 | 0.92 | 0.75 |
| **MLP** | 0.51 | 0.50 | 0.50 | 0.55 | 0.50 | 0.57 | 0.50 |
| **Naive Bayes** | 0.63 | 0.51 | 0.50 | 0.57 | 0.60 | 0.65 | 0.50 |

Table 7.2: 1-gram analysis AUC scores.

| AUC Scores | | | | | | | |
|---|---|---|---|---|---|---|---|
| **Classifier** | **Adware** | **Backdoor** | **Generic** | **Trojan** | **Virus** | **Worm** | **Unclassified** |
| **Random Forest** | 0.93 | 0.92 | 0.79 | 0.80 | 0.91 | 0.95 | 0.78 |
| **XGBoost** | 0.93 | 0.93 | 0.79 | 0.83 | 0.93 | 0.95 | 0.80 |
| **k-NN** | 0.91 | 0.85 | 0.74 | 0.71 | 0.79 | 0.92 | 0.77 |
| **MLP** | 0.78 | 0.74 | 0.51 | 0.58 | 0.77 | 0.85 | 0.52 |
| **Naive Bayes** | 0.73 | 0.70 | 0.57 | 0.63 | 0.66 | 0.85 | 0.65 |

Table 7.3: 2-gram analysis AUC scores.

| AUC Scores | | | | | | | |
|---|---|---|---|---|---|---|---|
| **Classifier** | **Adware** | **Backdoor** | **Generic** | **Trojan** | **Virus** | **Worm** | **Unclassified** |
| **Random Forest** | 0.93 | 0.92 | 0.78 | 0.80 | 0.90 | 0.95 | 0.79 |
| **XGBoost** | 0.93 | 0.91 | 0.78 | 0.81 | 0.90 | 0.95 | 0.80 |
| **k-NN** | 0.91 | 0.85 | 0.73 | 0.71 | 0.79 | 0.92 | 0.78 |
| **MLP** | 0.88 | 0.83 | 0.60 | 0.71 | 0.76 | 0.87 | 0.61 |
| **Naive Bayes** | 0.76 | 0.71 | 0.59 | 0.65 | 0.71 | 0.84 | 0.69 |

Table 7.4: 3-gram analysis AUC scores.

| | | | |
|---|---|---|---|
| FileSize | e_magic | e_cblp | e_cp |
| e_crlc | e_cparhdr | e_minalloc | e_maxalloc |
| e_ss | e_sp | e_csum | e_ip |
| e_cs | e_lfarlc | e_ovno | e_res |
| e_oemid | e_oeminfo | e_res2 | e_lfanew |
| Machine | NumberOfSections | CreationYear | PointerToSymbolTable |
| NumberOfSymbols | SizeOfOptionalHeader | Characteristics | Magic |
| MajorLinkerVersion | MinorLinkerVersion | SizeOfCode | SizeOfInitializedData |
| SizeOfUninitializedData | AddressOfEntryPoint | BaseOfCode | BaseOfData |
| ImageBase | SectionAlignment | FileAlignment | MajorOperatingSystemVersion |
| MinorOperatingSystemVersion | MajorImageVersion | MinorImageVersion | MajorSubsystemVersion |
| MinorSubsystemVersion | SizeOfImage | SizeOfHeaders | CheckSum |
| Subsystem | DllCharacteristics | SizeOfStackReserve | SizeOfStackCommit |
| SizeOfHeapReserve | SizeOfHeapCommit | LoaderFlags | NumberOfRvaAndSizes |
| rdata | data | rsrc | reloc |
| # other sections | rdata virtual size | data virtual size | reloc virtual size |
| rsrc virtual size | rdata entropy | data entropy | rsrc entropy |
| reloc entropy | KERNEL API COUNTS | USER API COUNTS | ADVAPI API COUNTS |
| SHELL API COUNTS | COMCTL API COUNTS | CRYPT API COUNTS | MSVCR API COUNTS |
| GDI API COUNTS | SHLWAPI API COUNTS | WS API COUNTS | WININET API COUNTS |
| WINHTTP API COUNTS | Imported libraries count | LoadLibrary | GetProcAddress |
| MessageBox | ShellExecute | IsDebuggerPresent | VirtualAlloc |
| CreateThread | CreateProcess | OpenProcess | RaiseException |
| CreateEvent | GetSystemInfo | GetComputerName | SetWindowsHook |
| WriteProcessMemory | GetTickCount | Sleep | GetDiskFreeSpace |
| SetThreadContext | CreateRemoteThread | GetVersion | GetProcessHeap |
| GetUserName | ExitProcess | CorExeMain | WaitForSingleObject |
| GetStartupInfo | GetKeyboard | SetUnhandledExceptionFilter | HttpSendRequest |
| HttpQueryInfo | | | |

Table 7.5: All header features.

| Feature | Importance |
|---|---|
| NumberOfSections | 0.246286 |
| Subsystem | 0.061335 |
| .data entropy | 0.054291 |
| SizeOfInitializedData | 0.046654 |
| HttpQueryInfo | 0.041567 |
| SizeOfHeaders | 0.040972 |
| Magic | 0.037701 |
| MinorSubsystemVersion | 0.036462 |
| .rdata entropy | 0.032309 |
| SizeOfUninitializedData | 0.030016 |
| SizeOfStackCommit | 0.029206 |
| BaseOfData | 0.021445 |
| MinorLinkerVersion | 0.021280 |
| SizeOfCode | 0.018719 |
| .reloc presence | 0.018187 |
| WINHTTP | 0.017894 |
| DllCharacteristics | 0.015969 |
| .rsrc entropy | 0.014974 |
| e_magic | 0.013570 |
| SizeOfOptionalHeader | 0.012410 |
| Machine | 0.012206 |
| GetKeyboard | 0.012156 |
| .data VirtualSize | 0.010271 |
| LoadLibrary | 0.010053 |
| .idata entropy | 0.009229 |

Table 7.6: Top 25 header features.

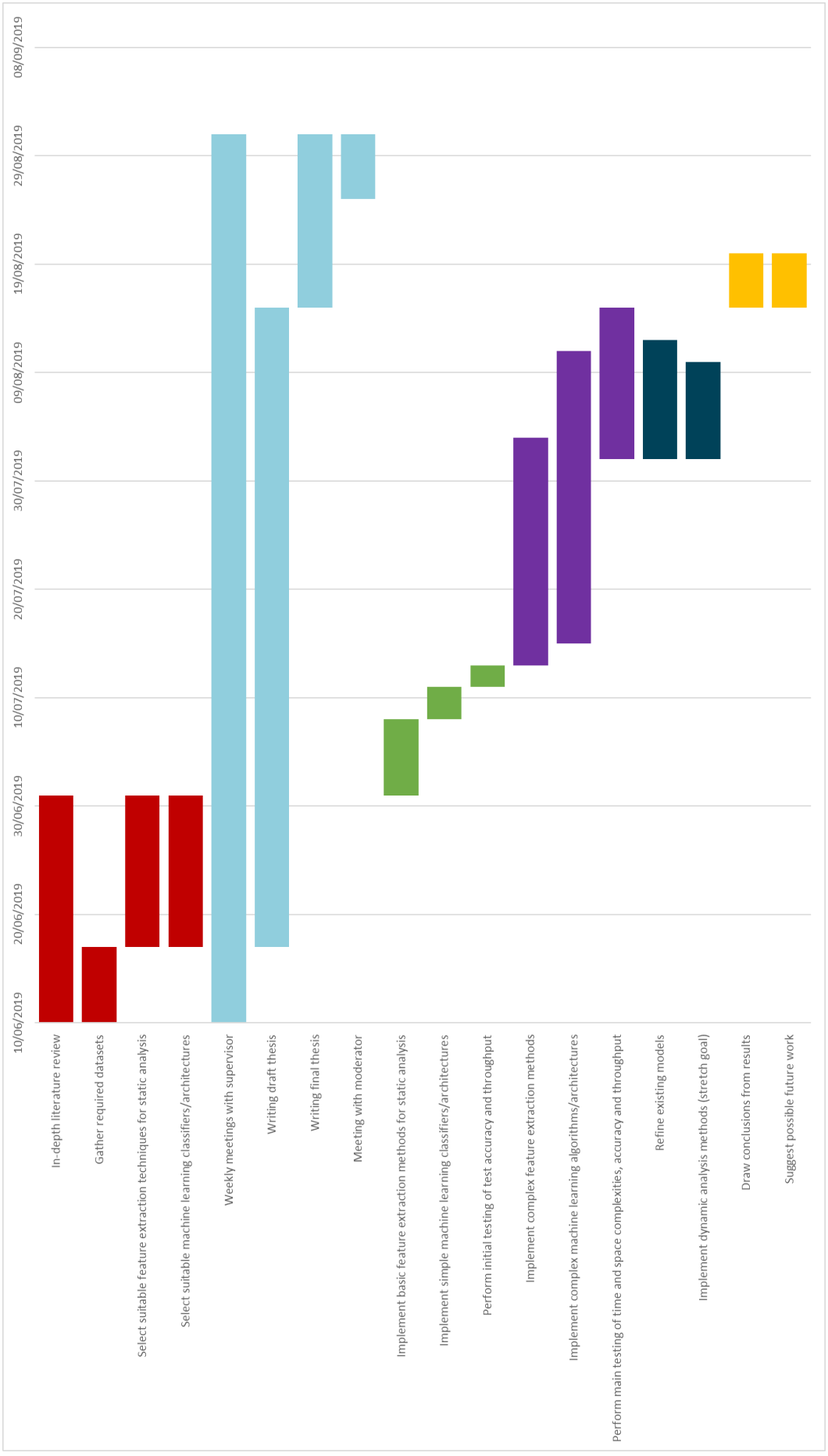



Figure 7.1: Original Gantt chart.